FASTRTPSGEN v1.0.0

User Manual

eProsima
Proyectos y Sistemas de Mantenimiento SL
Ronda del poniente 16 – Bajo K
28760 Tres Cantos Madrid
Tel: + 34 91 804 34 48
info@eProsima.com – www.eProsima.com

Trademarks

eProsima is a trademark of *Proyectos y Sistemas de Mantenimiento SL.* All other trademarks used in this document are the property of their respective owners.

License

eProsima Fast RTPS is licensed under the terms described in the FASTRTPS_LIBRARY_LICENSE.txt file included in this distribution.

Technical Support

Phone: +34 91 804 34 48

Email: Support@eProsima.com

Contents

1. Introduction....1

2. Building your publisher/subscriber code....2

3. Defining your data types....4

3.1. Basic types....4

3.2. Arrays....4

3.3. Sequences....5

3.4. Structures....5

3.5. Unions....6

3.6. Enumerations....7

3.7. Keyed Types....8

4. HelloWorld example....9

4.1. Writing the IDL file....9

4.2. Generating specific code....9

4.3. Finishing the generated code....9

4.3.1. Publisher behavior....10

4.3.2. Subscriber behavior....10

4.4. Building and executing....10

Figure List

Table List

1 Introduction.

eProsima FASTRTPSGEN is a Java application that generates source code using the data types defined in an IDL file. This generated source code can be used in your applications in order to publish and subscribe to a topic of your defined type.

To declare your structured data, you have to use IDL (Interface Definition Language) format. IDL is a specification language, made by OMG (Object Management Group), which describes an interface in a language-independent way, enabling communication between software components that do not share the same language.

eProsima FASTRTPSGEN is a tool that reads IDL files and parses a subset of the OMG IDL specification to generate serialization source code. This subset includes the data type descriptions included in section 3. The rest of the file content is ignored.

eProsima FASTRTPSGEN generated source code uses a relevant component: a C++11 library that provides a serialization mechanism. In this case, as indicated by the RTPS specification document, the serialization mechanism used in CDR. The standard CDR (Common Data Representation)[1] is a transfer syntax low-level representation for transfer between agents, mapping from data types defined in OMG IDL to byte streams.

One of the main features of *eProsima FASTRTPSGEN* is to avoid the users the trouble of knowing anything about serialization or deserialization procedures. It also provides a first implementation of a publisher and a subscriber using eProsima RTPS library.

This document presents a brief guide of how to use eProsima FASTRTPSGEN. The structure of the document is presented below:

- You begin learning how to use the application to create your source code in section 2 (Building your publisher/subscriber code).
- All supported data types by *eProsima FASTRTPSGEN* are listed in section 3 (Defining your data types). This section also shows how to define these data types using the IDL definition language.
- In section 4 (HelloWorld example) a complete HelloWorld example is shown.

This release of *eProsima FASTRTPSGEN* only supports the generation of source code in C++11. Forthcoming releases will integrate more programming languages.

1 Visit http://www.omg.org/cgi-bin/doc?formal/02-06-51 for more information about CDR serialization standard

2 Building your publisher/subscriber code

This section guides you through the usage of this Java application and briefly describes the generated files.

The Java application can be executed using the following script:

In Windows:

```
> fastrtpsgen.bat
```

Note: The script will be found by Windows if its location has been set in the PATH environment variable. This can be done during the installation process. More information can be found in the Installation Manual.

In Linux:

```
$ fastrtpsgen.sh
```

eProsima FASTRTPSGEN Java application is capable of processing several IDL files. The expected argument line is:

```
fastrtpsgen [<options>] <IDL file> [<IDL file> ...]
```

where the available options are:

Option	Description
-help	Shows the help information.
-version	Shows the current version of *eProsima FASTRTPSGEN*.
-d <directory>	Output directory where the generated files are created.
-example <platform>	Generates an example and a solution to compile the generated source code for a specific platform. The help command shows the supported platforms.
-replace	Replaces the generated source code files whether they exist.

For each processed IDL file *eProsima FASTRTPSGEN* tool generates nine source code files.

The next table summarizes the content of each generated file, supposing a file named Example.idl is passed to the application:

File	Description
Example.h /.cpp	C++ Header and source file containing the user data types mapped to C++11 programming language, as well as serialization and deserialization methods for the types.
ExamplePublisher.h/.cpp	C++ code where a publisher is created and used to send the user defined types.
ExampleSubscriber.h/.cpp	C++ code where a subscriber is created and used to receive the user defined types.
ExamplePubSubType.h/.cpp	C++ code that uses the serialization and deserialization methods defined for the types to serialize the data into the message that the publisher sends (or the subscriber receives). It also defines the method needed to obtain the key from the data type.
ExamplePubSubMain.cpp	Main file of the example. It parses the argument (publisher or subscriber) and executes the selected side accordingly.

3 Defining your data types

You have to define your data types using IDL format. Not all IDL features are supported in this version of *eProsima FASTRTPSGEN*. This section shows you what IDL types are supported and how *eProsima FASTRTPSGEN* maps them in C++11 programming language.

For each IDL file parsed by *eProsima FASTRTPSGEN* two source code files are generated. They contain the C++11 mapping of your IDL types. For example, from an IDL file named *Foo.idl*, *eProsima FASTRTPSGEN* tool generates the files *Foo.h* and *Foo.cpp* that contain the C++11 mapping of the defined IDL types.

Next subsections introduce the supported IDL types.

3.1 Basic types

The following table shows the basic IDL types supported by *eProsima FASTRTPSGEN* and how they are mapped to C++11.

IDL	C++11
`char vartype;`	`char vartype;`
`octet vartype;`	`uint8_t vartype;`
`short vartype;`	`int16_t vartype;`
`unsigned short vartype;`	`uint16_t vartype;`
`long vartype;`	`int32_t vartype;`
`unsigned long vartype;`	`uint32_t vartype;`
`long long vartype;`	`int64_t vartype;`
`unsigned long long vartype;`	`uint64_t vartype;`
`float vartype;`	`float vartype;`
`double vartype;`	`double vartype;`
`boolean vartype;`	`bool vartype;`
`string vartype;`	`std::string vartype;`

3.2 Arrays

eProsima FASTRTPSGEN supports arrays and multidimensional arrays. An array is mapped to the STD array container. The following table shows the supported array types and how they are mapped.

IDL type	C++11 type
`char arraytype[5];`	`std::array<char, 5> arraytype;`
`octet arraytype[5];`	`std::array<uint8_t, 5> arraytype;`
`short arraytype[5];`	`std::array<int16_t, 5> arraytype;`
`unsigned short arraytype[5];`	`std::array<uint16_t, 5> arraytype;`
`long arraytype[5];`	`std::array<int32_t, 5> arraytype;`

`unsigned long arraytype[5];`	`std::array<uint32_t, 5> arraytype;`
`long long arraytype[5];`	`std::array<int64_t, 5> arraytype;`
`unsigned long long arraytype[5];`	`std::array<uint64_t, 5> arraytype;`
`float arraytype[5];`	`std::array<float, 5> arraytype;`
`double arraytype[5];`	`std::array<double, 5> arraytype;`

3.3 Sequences

eProsima FASTRTPSGEN supports sequences. A sequence is mapped to the STD vector container. The following table shows the supported sequence types and how they are mapped.

IDL	C++11
`sequence<char> sequencetype;`	`std::vector<char> sequencetype;`
`sequence<octet> sequencetype;`	`std::vector<uint8_t> sequencetype;`
`sequence<short> sequencetype;`	`std::vector<int16_t> sequencetype;`
`sequence<unsigned short> sequencetype;`	`std::vector<uint16_t> sequencetype;`
`sequence<long> sequencetype;`	`std::vector<int32_t> sequencetype;`
`sequence<unsigned long> sequencetype;`	`std::vector<uint32_t> sequencetype;`
`sequence<long long> sequencetype;`	`std::vector<int64_t> sequencetype;`
`sequence<unsigned long long> sequencetype;`	`std::vector<uint64_t> sequencetype;`
`sequence<float> sequencetype;`	`std::vector<float> sequencetype;`
`sequence<double> sequencetype;`	`std::vector<double> sequencetype;`

3.4 Structures

In IDL, a structure is defined as a set of members, each one with its own type. An IDL structure type is mapped as a C++ class, while each member is mapped as a class attribute. A set of methods to get and set the attribute is also created. The following table shows and example of an IDL structure type and its mapping to the C++11 programming language.

IDL	C++11

struct Structure
{
octet octet_value;
long long_value;
string string_value;
};
	class Structure
	{
	public:
	Structure();
	~Structure();
	Structure(const Structure &x);
	Structure(Structure &&x);
	Structure& operator=(const Structure &x);
	Structure& operator=(Structure &&x);
	
	void octet_value(uint8_t _octet_value);
	uint8_t octet_value() const;
	uint8_t& octet_value();
	void long_value(int64_t _long_value);
	int64_t long_value() const;
	int64_t& long_value();
	void string_value(const std::string
	&_string_value);
	void string_value(std::string &&_string_value);
	const std::string& string_value() const;
	std::string& string_value();
	
	private:
	uint8_t m_octet_value;
	int64_t m_long_value;
	std::string m_string_value;
	};

3.5 Unions

In IDL, a union is defined as a sequence of members with their own types and a discriminant that specifies which member is in use. An IDL union type is mapped as a C++ class with access functions to the union members and the discriminant. The following table shows an example of IDL union type and its mapping to the C++11 programming language.

IDL	C++11

union Union switch(long)
{
case 1:
octet octet_value;
case 2:
long long_value;
case 3:
string string_value;
};
	class Union
	{
	public:
	Union();
	~Union();
	Union(const Union &x);
	Union(Union &&x);
	Union& operator=(const Union &x);
	Union& operator=(Union &&x);

```
    void d(int32t __d);
    int32_t _d() const;
    int32_t& _d();

    void octet_value(uint8_t _octet_value);
    uint8_t octet_value() const;
    uint8_t& octet_value();
    void long_value(int64_t _long_value);
    int64_t long_value() const;
    int64_t& long_value();
    void string_value(const std::string
       &_string_value);
    void string_value(std:: string &&_string_value);
    const std::string& string_value() const;
    std::string& string_value();

private:
    int32_t m__d;
    uint8_t m_octet_value;
    int64_t m_long_value;
    std::string m_string_value;
};
```

3.6 Enumerations

An enumeration in IDL format is a collection of identifiers that have a numeric value associated. An IDL enumeration type is mapped directly to the corresponding C++11 enumeration definition. The following table shows an example of an IDL enumeration type and its mapping to the C++11 programming language.

IDL	C++11

`enum Enumeration`
`{`
`  RED,`
`  GREEN,`
`  BLUE`
`};`
	`enum Enumeration : uint32_t`
	`{`
	`  RED,`
	`  GREEN,`
	`  BLUE`
	`};`

3.7 Keyed Types

In order to use keyed topics the user should define some key members inside the structure. This is achieved by writting “@Key” before the members of the structure you want to use as keys.

For example in the following IDL file:

```
struct MyType
{
    @Key long id;
    @Key string type;
    long positionX;
    long positionY;
};
```

the fields “id” and “type” would be the keys in the type.

The FASTRTPSGEN application automatically detects these tags and correctly generates the serialization methods for the key generation function in TopicDataType (getKey). This function will obtain the 128 MD5 digest of the big endian serialization of the Key Members.

4 HelloWorld example

This section explains how to write a HelloWorld example step by step. This example has a simple structure with a string and a long id to transmit between the publisher and the subscriber;

4.1 Writing the IDL file

The first step is to write an IDL file containing the structures you want to serialize/deserialize. In this case, it is a simple structure:

```
struct HelloWorld
{
    @Key long id;
    string message;
};
```

Save this file as *HelloWorld.idl*.

4.2 Generating specific code

Open a command prompt and go to the directory containing *HelloWorld.idl*.

If you are running this example in Windows, execute the following line:

```
fastrtpsgen -example x64Win64VS2010 HelloWorld.idl
```

If you are running it in Linux, execute this one:

```
fastrtpsgen -example x64Linux2.6gcc HelloWorld.idl
```

Note that if you are running *eProsima FASTRTPSGEN* in a 32-bit operating system you have to use *-example i86Win32VS2010* or *-example i86Linux2.6gcc* instead.

This command generates your serialization code and some project files designed to build your HelloWorld example. In Windows, it generates a Visual Studio 2010 solution, named solution-x64Win64VS2010*.sln*. This solution has a single project called *HelloWorld*, with all the needed code already included. In Linux, on the other hand, it generates a makefile.

4.3 Finishing the generated code

To finish the behaviour of the publisher and the subscriber you need to edit the run method in the *HelloWorldPublisher.cxx* file and the *onNewDataMessage* method of the listener of the subscriber in the HelloWorldSubscriber.cxx file.

4.3.1 Publisher behavior

```
void HelloWorldPublisher::run()
{
	while(m_listener.n_matched == 0)
	{
		eClock::my_sleep(250); // Sleep 250 ms
	}
	// Publication code
	HelloWorld st;
	/* Modify the values of your structure here */
	mp_publisher->write(&st);
}
```

4.3.2 Subscriber behavior

```
void exampleSubscriber::SubListener::onNewDataMessage()
{
	if(mp_sub!=NULL)
	{
		// Take data
		HelloWorld st;
		if(mp_sub->takeNextData(&st, &m_info))
		{
			if(m_info.sampleKind == ALIVE)
			{
			 // Print your structure data here.
			}
	}
}
```

4.4 Building and executing

To build your code using Visual Studio 2010, make sure you are in the Debug (or Release) profile, and then build it (F7). Now go to `<example_dir>\bin\x64Win64VS2010` directory and execute HelloWorldExample.exe with the correct argument (either *publisher* or *subscriber*).

To build your code in Linux use this command:

```
make -f makefile_x64Linux2.6gcc
```

In both scenarios you should start the same executable in two different terminals passing publisher and subscriber to each of them.